TEXAS INSTRUMENTS

20450 Century Boulevard
Germantown, MD 20874
Fax: (301) 515-7954

SRIO LLD

Software Design Specification (SDS)

Revision A

Revision Record	
Document Title: **Software Design Specification**	
Revision	Description of Change
A	1. Initial Release – Code Drop 1.0.0.0 2. Added Multi-core enhancements – Code Drop 1.0.0.1 3. New CPPI/QMSS Specification upgrades – Code Drop 1.0.0.2 4. Raw Socket Support, Extended configuration, Transmit Completion Queue has been removed, Receive ISR support has been added, Device Initialization support added – Code Drop 1.0.0.3 5. Cache Hooks, OSAL API Extended, Device Layer is now outside the context of the SRIO driver – Code Drop 1.0.0.8 6. DIO sockets – Code Drop 1.0.0.9 7. DIO socket extensions – Code Drop 1.0.1

Note: Be sure the Revision of this document matches the QRSA record Revision letter. The revision letter increments only upon approval via the Quality Record System.

TABLE OF CONTENTS

1 Scope

This document describes the functionality, architecture, and operation of the SRIO Low Level Driver.

2 References

The following references are related to the feature described in this document and shall be consulted as necessary.

No	Referenced Document	Control Number	Description
1	SRIO User Guide	Version 0.0.7	SRIO User Guide
2	CPPI User Guide	Version 0.5.7	CPPI User Guide
3	SRIO Driver Documentation		The SRIO Driver API is generated by DOXYGEN and is located in the SRIO package under the "`docs`" directory in CHM format.

Table 1. Referenced Materials

3 Definitions

Acronym	Description
API	Application Programming Interface
DSP	Digital Signal Processor
RIO	Rapid IO
BSD	Berkeley Software Distribution

Table 2. Definitions

4 Overview

The *RapidIO*^{TM} is a non-proprietary high-bandwidth system level interconnects. It is intended to offer Gigabyte per second performance levels for chip-to-chip and board-to-board communication. Its layered architecture, allows a highly scalable interconnect capable of future enhancements.

The *RapidIO* is defined as a 3-layer architectural hierarchy:-

1. **Logical Layer**

   The Logical layer specifies the protocol, including packet formats, which are needed by end points to process transactions.

2. **Transport Layer**

   The Transport layer defines addressing schemes to correctly route information packets within a system.

3. **Physical Layer**

   The Physical layer specification contains the device level interface information such as electrical characteristics, error management, and basic flow control.

The following communication protocols are supported:-

1. **Direct I/O**

   The *RapidIO* packet contains the specific address where the data should be stored or read in the destination device. This means that the source device must have detailed knowledge of the available memory space within the destination device.

2. **Type 9**

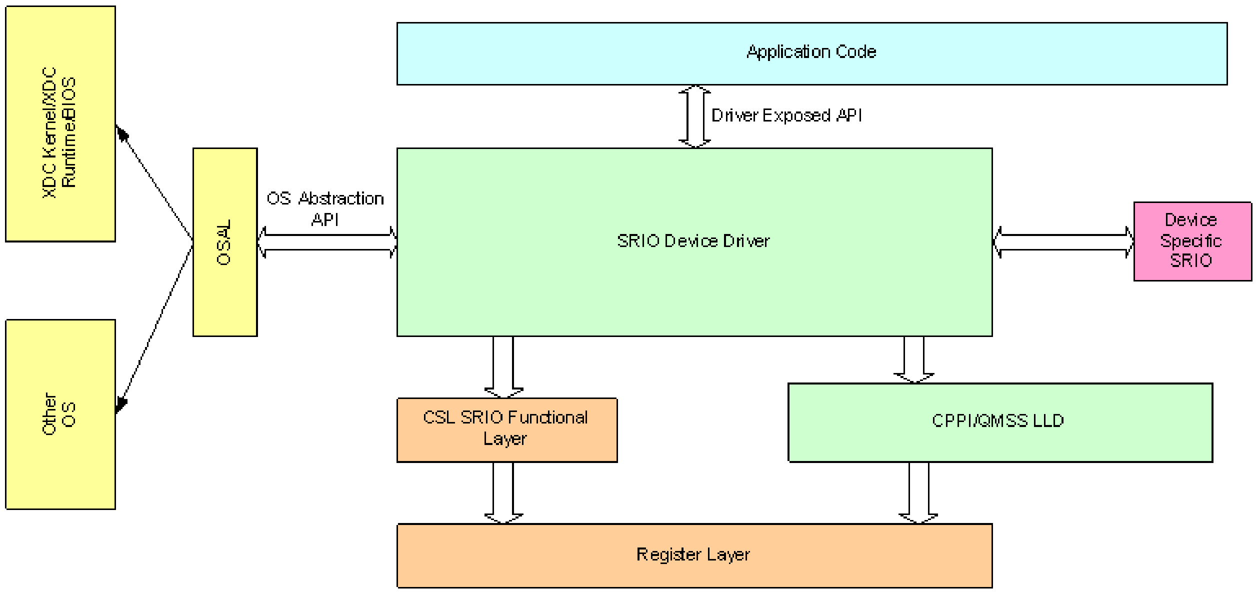
   A destination address is not specified. Instead the Stream ID is used to map the request to a specific memory region by the local (destination) device. TI IP additionally uses Class of Service (COS) as well to further classify the mapping of a request to a memory region.

3. **Type 11**

   A destination address is not specified, instead, a mailbox identifier is used within the *RapidIO* packet. The mailbox is controlled and mapped to memory by the local (destination) device.

The SRIO LLD driver provides a well defined API layer which allows applications to use the *RapidIO* peripheral to send and receive data using either of the above mentioned communication protocols. The next sections document the design details of the SRIO LLD driver.

The following is an architecture figure which showcases the SRIO LLD Driver architecture:-



The figure illustrates the following key components:-

1. **SRIO Device Driver**

   This is the core SRIO device driver. The device driver exposes a set of well defined API which is used by the application layer to send and receive data via the *RapidIO* peripheral. The driver also exposes a set of well defined OS abstraction API which is used to ensure that the driver is OS independent and portable. The SRIO driver uses the CSL SRIO functional layer for all SRIO MMR accesses. The SRIO driver also interfaces with the CPPI and QMSS libraries to be support the Type9 and Type11 protocols.

2. **Device Specific SRIO layer**

   This layer implements a well defined interface which allows the core SRIO driver to be ported on any device which has the same SRIO IP block. This layer changes for every device.

3. **Application Code**

   This is the user of the driver and its interface with the driver is through the well defined API set. Applications users use the driver API's to send and receive data via the *RapidIO* peripheral

4. **Operating System Abstraction Layer (OSAL)**

   The SRIO LLD is OS independent and exposes all the operating system callouts via this OSAL layer.

5. **CSL Functional Layer**

   The SRIO LLD driver uses the CSL SRIO functional layer to program the device IP by accessing the MMR (Memory Mapped Registers).

6. **Register Layer**

   The register layer is the IP block memory mapped registers which are generated by the IP owner. The SRIO LLD driver does not directly access the MMR registers but uses the SRIO CSL Functional layer for this purpose.

The sections later in the document address the design of each of the above mentioned components in more detail

5 Design

The SRIO driver is responsible for the following:-

1. Configurable CPPI Descriptor & Queue Management
2. Providing a well defined API to interface with the applications
3. Well defined operating system adaptation layer API

The next couple of sections document each of the above mentioned responsibilities in greater detail:

5.1 SRIO Peripheral Configuration

The SRIO driver provides a sample implementation sequence which initializes the SRIO IP block. This implementation is **sample only** and application developers are recommended to modify it as deemed fit.

The initialization sequence is **not** a part of the SRIO driver library. This was done because the SRIO Device Initialization sequence has to be modified and customized by application developers. If the initialization sequence was a part of the SRIO driver then it would require the drivers to be rebuilt. Moving this API outside the driver realm solves this issue. Application developers now need to ensure that they call the SRIO Device Initialization before they call the SRIO Driver Init API.

The SRIO Device initialization API is implemented as a sample prototype:

```
int32_t SrioDevice_init (void)
```

Application developers need to ensure that they call the SRIO Device Initialization before they call any SRIO Driver API. Failure to do so will result in unpredictable behaviors.

5.2 SRIO Driver Initialization

The SRIO Driver initialization API needs to be called only once and it initializes the internal driver data structures. The initialization API should only be called after the SRIO Device layer has been properly initialized.

The following API [3] is used to initialize the SRIO Driver.

```
int32_t Srio_init (void)
```

The function returns 0 on success indicating that the SRIO driver internal data structures have been initialized correctly and returns a negative value on error.

5.3 CPPI Descriptor & Queue Management

To ensure that the driver executes across multiple cores and to satisfy the requirements of multiple applications which use the driver; the driver exposes a concept called "*Driver Instances*".

Each driver instance needs to configure various CPPI & QMSS parameters before it can be used. These configurations are exposed to the applications as follows:

- ***Application Managed Configuration***
  In this configuration the application is exposed to all the lower level CPPI and QMSS details. This is aimed primarily for advanced use cases where lower level control on various aspects of the SRIO driver is required by the application. The configuration exposes the CPPI Receive Flow configuration, Accumulator programming ability etc. Since the configuration is aimed for advanced use cases only RAW sockets will work on this driver instance.
- ***Driver Managed Configuration***
  In this configuration the application is encapsulated from a majority of the lower level CPPI and QMSS details. This configuration was aimed for use cases where applications would use the SRIO driver for data transfers without getting into the lower level details of the QMSS and CPPI. This configuration works only with Normal sockets which provide a higher level abstract interface.

The following API [3] is used to create and configure a SRIO driver instance

```
Srio_DrvHandle Srio_start (Srio_DrvConfig* ptrCfg)
```

The function returns a handle to the driver instance. All other API's in the SRIO driver can only be used after a driver instance has been created.

5.4 Driver API

As mentioned above the *Rapid IO* peripheral supports the following communication protocols:-

1. Direct IO
2. Type9
3. Type11

The SRIO driver was designed such that the library could handle all communication protocols. The design was based on the well known BSD socket API which achieves the same by supporting the networking socket API over different protocol families i.e. IPv4, IPv6, IPX etc.

From an application perspective; a SRIO socket is basically an endpoint which can be used to send and receive data. The SRIO socket abstracts the inner details of the SRIO peripheral from the end user.

5.4.1 Socket Open

Application developers open SRIO sockets by using the following API [3]:

```
Srio_SockHandle SRIO_sockOpen (
      Srio_DrvHandle  hSrioDriver,
      Srio_SocketType type
      Bool            isBlocking
      );
```

Sockets can only be opened on a specific driver instance. The API returns the `Srio_SockHandle` which is an abstract socket handle exposed to the applications. The application should use this socket handle for all further socket operations.

Applications need to specify the socket communication protocol (DIO, Type9, Type11 etc) to be used. For Type9 and Type11 communication protocols operate in either of the following modes:-

- **Normal Mode**
  In this mode sockets operate using data payload for all send and receive operations. Normal mode sockets do not support sending & receiving scattered buffers.
- **Raw Mode**
  In this mode sockets operate using buffer descriptors for all send and receive operations. These sockets support sending & receiving scattered buffers.

For DIO sockets the mode parameter is a don't care.

Sockets can also have either of the following characteristic:-

- **Blocking**
  These sockets will cause the application thread which is calling the SRIO Driver exposed API to block on a semaphore until the specified condition is removed. *For example*: If an application calls the receive API to read data from a socket but none is available; then in that case a blocking socket will cause the application to block. Once data is received on the specified socket; the application is woken up.
- **Non-Blocking**
  A non blocking socket will simply return an error or 0 indicating that the specified condition could not be met. It is the responsibility now of the application to determine the action which is required to be taken.

Each socket internally maintains a queue of received packets; the reasoning behind this is that packets will be received by the drivers (say in ISR context) but the applications might read from the sockets as per the application requirements. The received data needs to be stored in the order in which it was received.

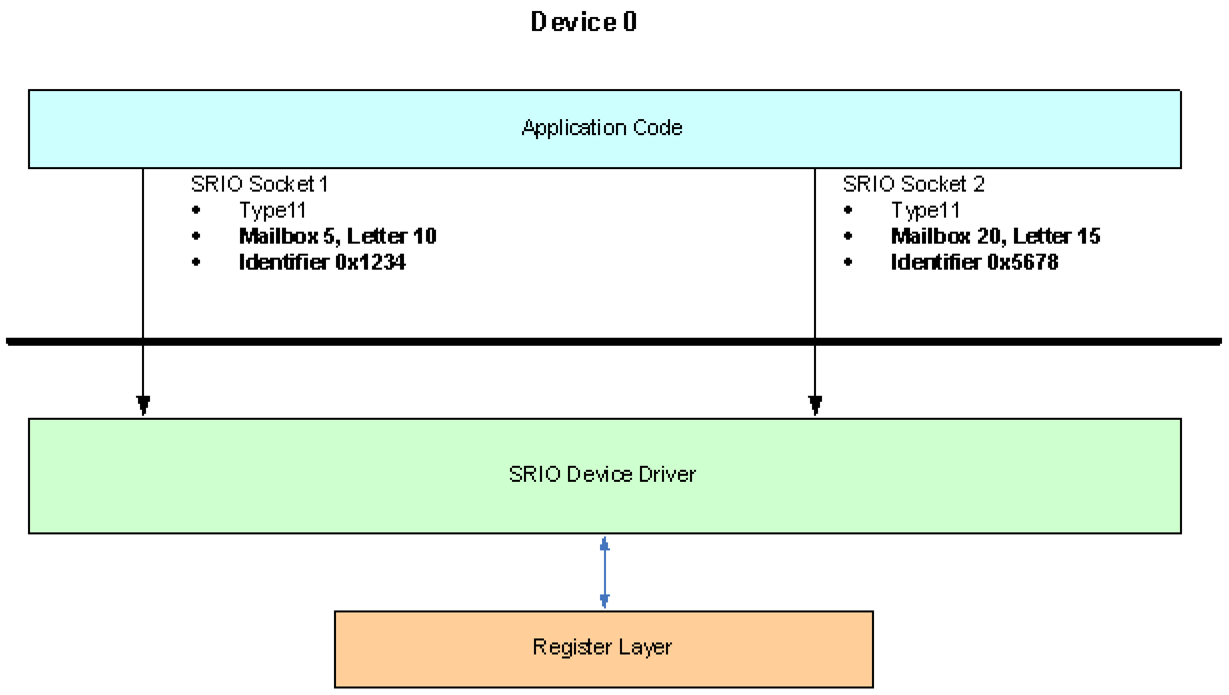
The default depth of the socket receive queue is controlled by the following definition

```
#define DEFAULT_SRIO_MAX_PENDING_PACKETS    5
```

Applications can change the default configuration but will need to rebuild the driver libraries. The SRIO Driver also exposes a run time configuration which allows applications to change the limits on a per socket basis.

5.4.2 Socket Bind

Once the SRIO socket has been opened; it new to be bound to some local "*communication protocol*" specific properties. This is required to differentiate between multiple sockets opened on the same device. The following figure illustrates this:-



In this scenario; the application needs to open multiple sockets belonging to say the socket type i.e. Type11; but the properties of the sockets are different. This is known as a local binding. All sockets need to be *bound* before they can be used for sending/receiving data. The following API [3] is used to perform this:

```
Int32 SRIO_sockBind (Srio_SockHandle    srioSock,
                     Srio_SockAddrInfo* ptr_addrInfo);
```

The binding information is socket type specific. The `Srio_SockAddrInfo` union encapsulates this information for each type.

In networking this is similar for example to a web-server binding itself to port 80. The web server listens for connections coming only port 80 and processes them accordingly.

Internally each SRIO socket is mapped to a binding entry in the SRIO IP block. This one-one mapping imposes an upper limit to the maximum number of Type9/Type11 and DIO sockets which can be opened.

For DIO sockets the binding maps the socket to an LSU entry. There are 8 LSU entries which are available in the SRIO IP block. During binding for DIO sockets the following fields [3] in the LSU transfer data structure are pre-populated:-

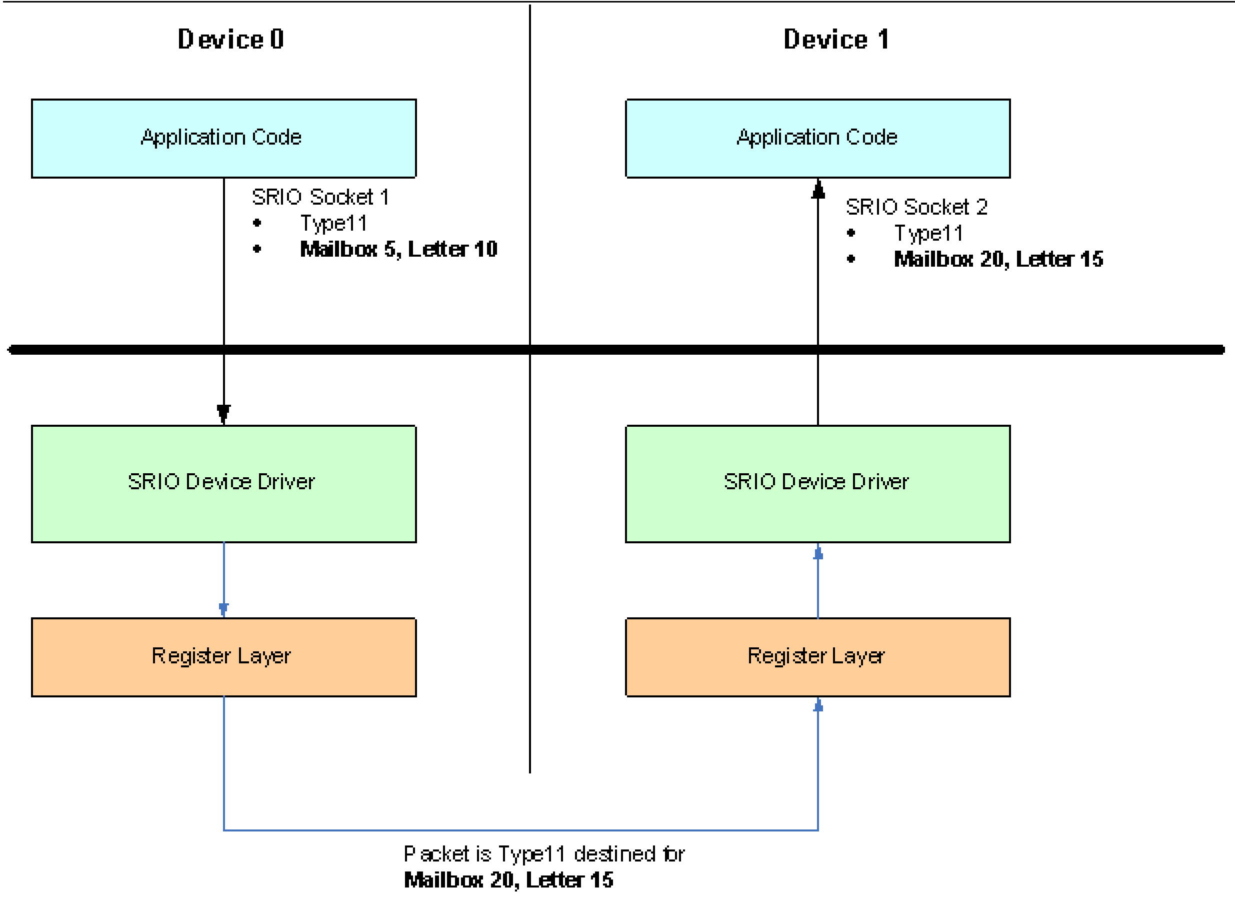
- doorbellValid
- intrRequest
- supInt
- xambs
- priority
- outPortID
- idSize
- srcIDMap
- hopCount
- doorbellInfo

The remaining fields used to initiate an LSU transfer are configured in the socket send. This is done to improve performance on DIO sockets since most of the information is configured and stored upfront in the LSU transfer during socket bind.

For DIO sockets a doorbell transfer LSU structure is also pre-populated with the appropriate information during socket bind. All the fields in the LSU transfer except Destination id, Destination id size (16 bit or 8bit) and the doorbell information are pre-populated. The remaining fields are only populated while sending the doorbell. This is done to improve performance while sending doorbells.

5.4.3 Socket Send

Once the socket has been opened and bound it can now send and receive data. To be able to send data an application should know the address information of the peer to which it wishes to send data. The following figure illustrates this:-



In this scenario; if SRIO socket1 wishes to send a packet to SRIO socket2; it needs to know the information and properties of the destination socket i.e. the local binding information of socket2.

The following API [3] is used to perform the send operation:

```
Int32 Srio_sockSend (Srio_SockHandle    srioSock,
                     Srio_DrvBuffer     hDrvBuffer
                     UInt32             num_bytes,
                     Srio_SockAddrInfo* to);
```

The API initializes the SRIO Protocol Specific Information [3] and then pushes the buffer descriptor into the transmit queue.

The API behavior is dependent on the socket type and is explained in more detail in the next sections:

5.4.3.1 Normal Mode

For Normal mode sockets which operate at data payload level; applications need to allocate and get a SRIO Driver Buffer (`Srio_DrvBuffer`). This implies that the driver instance should be configured appropriately. The following API [3] is used to perform the allocation operation:

```
Srio_DrvBuffer Srio_allocTransmitBuffer (Srio_DrvHandle hSrioDrv,
                                         UInt8** ptrData,
                                         UInt32* bufferLen)
```

The function returns a pointer to the data buffer (`ptrData`); which can be used along with the length of the allocated buffer (`bufferLen`). Applications can now use the data buffer and populate it with the payload to be sent.

Once the application is done with populating the payload it uses the send API to transmit the data over the SRIO IP block. The argument '`num_bytes`' in the SRIO Socket send API should be set to the size of the data payload being transmitted.

The allocated buffer is automatically cleaned up if the send API was successful.

On an error; it is the responsibility of the application to use the following API [3] to clean up

```
Void Srio_freeTransmitBuffer(Srio_DrvHandle hSrioDrv,
                             Srio_DrvBuffer hDrvBuffer)
```

The reason why this schema was adopted is because the data buffers are allocated from within the driver pool. Configuration of the transmit buffer descriptors with the return queue set to the Transmit free queue itself allows recycling the data buffers with no host intervention (Interrupts etc)

Another design pattern here would be to take a transmit buffer payload directly from the application itself; link it with the buffer descriptor and send it across. This however is costly because it implies that the SRIO driver now needs to monitor Transmit Completion Interrupts which then need to be hooked to the application memory management cleanup API. This has a performance penalty.

5.4.3.2 Raw Mode

Raw Mode sockets since they operate at buffer descriptor. This implies that it is the responsibility of the application to do the following:-

- Allocate a Buffer Descriptor
- Ensure that the return queue in the buffer descriptor is correctly configured.
- Link it with the valid data buffer with the application payload.

- Ensure that the data buffer lengths are correctly configured.

The argument ‘`num_bytes`’ in the SRIO Socket send API should be set to the size of the buffer descriptor which is being transmitted.

Once the send operation is complete the transmit buffer descriptor is automatically recycled back to the return queue specified in the buffer descriptor.

5.4.3.3 DIO sockets

For DIO sockets a send operation implies in the following steps:

1. Populating the missing LSU Transfer information
    a. Destination Address (MSB & LSB) of the remote device on which the operation is to be performed.
    b. Source Address of the local device on which the specified operation is to be performed.
    c. Number of bytes of data transfer which is being performed.
    d. FTYPE and TTYPE which are SRIO commands which describe the type of operation.
    e. Destination id of the remote device.
2. Ensure that the LSU shadow registers are not full and there is space available to write the LSU transfer information.
3. Get the LSU context and Transaction information
4. Initiate the LSU transfer

The behavior of the SRIO sockets after the transfer is initiated changes depending on the socket type:

Blocking sockets:

In the case of blocking DIO sockets; once the LSU transfer has been initiated the SRIO driver will wait for completion by continuously polling the LSU Status. The function returns the completion code. Please refer to the SRIO User Guide [3] for more information on the values.

Non-Blocking sockets:

In the case of non-blocking sockets the SRIO driver would simply return from the send API once the LSU transfer is initiated. Application developers are responsible for checking the status of the completion by using the `Srio_getSockOpt` API

5.4.3.4 Sending Doorbell

DIO sockets are the only sockets capable of sending doorbells. To send a doorbell application need to configure the doorbell register and bit which is to be sent across and configure this in the

SRIO Driver Buffer (`Srio_DrvBuffer`). The following macro [3] is provided to help construct the doorbell information

```
#define SRIO_SET_DBELL_INFO(DBELL_REG, DBELL_BIT) CSL_FMKR(31, 16, DBELL_REG)|
                                                  CSL_FMKR(15,  0, DBELL_BIT)
```

This doorbell information is then populated in the `hDrvBuffer` parameter in the `Srio_sockSend` API. Please ensure that the FTYPE parameter is configured for DOORBELL.

5.4.4 Receive Interrupt/ Polling support

The SRIO Driver can operate in either interrupt or polling mode. The SRIO driver exposes the following API [3]

```
void Srio_rxCompletionIsr (Srio_DrvHandle hSrioDrv)
```

To handle DIO Doorbells the SRIO driver exposes the following API [3]

```
void Srio_dioCompletionIsr (Srio_DrvHandle hSrioDrv, uint8_t intDstDoorbell[])
```

The DIO Doorbell ISR exposes the parameter `intDstDoorbell` which is used to hold the index of the interrupt destination decode registers which need to be read to determine which doorbell interrupts are pending. Applications can determine the interrupt routing for the doorbell interrupts depending upon the configuration of the SRIO IP; since this configuration is outside the context of the SRIO driver; this information needs to be provided in the ISR.

In Interrupt Mode the application needs to install these ISR handlers with their OS interrupt dispatcher. Failure to so correctly will result in the SRIO driver not being able to receive any packets or handle DIO interrupts.

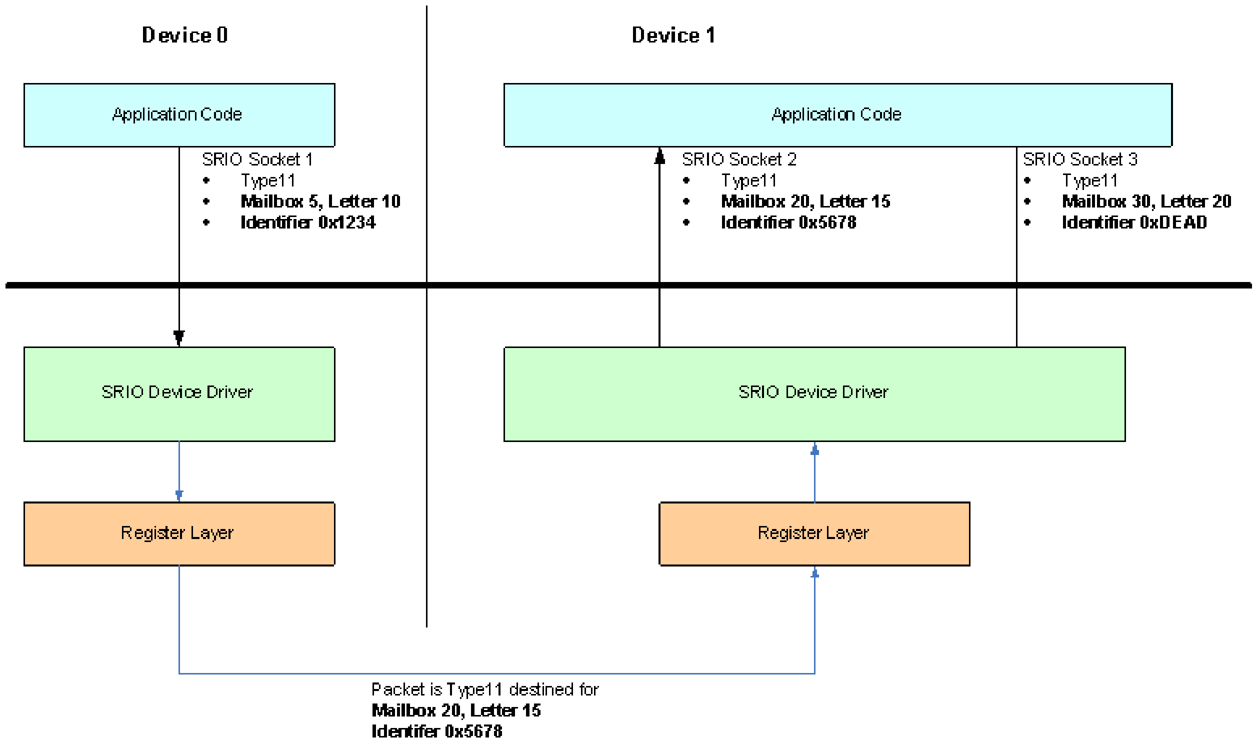
In Polled Mode the application is responsible for periodically these API.

The Receive completion ISR API does the following:-

1. Determines the Mode in which the driver instance is executing
   - Interrupt Mode
     Gets the descriptor on which data has been received by reading the accumulator list.
   - Polled Mode
     Gets the descriptor on which data has been received by reading from the receive completion queue.
2. Extracts the SRIO Protocol specific information from the received buffer descriptor

3. Find a socket which matches the protocol specific information with the local binding information of the socket.
4. Once a matching socket has been found; processing is socket specific
   - <u>Normal Mode</u>
     In the case of normal mode sockets; the data buffer from the received buffer descriptor is placed into the internal socket queues. A new data buffer is requested using the OSAL Memory allocation API attached to the descriptor & placed back into the receive queue.
   - <u>Raw Mode</u>
     For Raw mode sockets; the BD is placed into the internal socket queue as is.
5. Once the internal socket queue has been populated; if the socket is a blocking socket the semaphore associated with the socket is posted. This will wake up any application task which was waiting for data on that specific socket.

The following figure illustrates the above procedure:



In this scenario; Device1 has opened 2 sockets `Socket2` and `Socket3` each of which are bound to different "`socket type`" properties. The application on Device0; sends a packet to `Mailbox 20, Letter 15`. This causes an interrupt to be raised on Device1 which invokes the ISR API

The ISR API needs to now cycle through the socket database and determine the socket for which the data was received. It uses the protocol specific information of the received packet (`Mailbox 20, Letter 15 and Identifier 0x5678`) to do so. The API matches this to `Socket2` and the received packet is now placed into `Socket2` internal queue.

The DIO completion ISR does the following:-

1. Cycles through all the doorbell interrupt status register as specified by the input interrupt configuration
2. Determines which doorbell bits are pending and determines if there is a socket registered with the doorbell register and bit combination.
3. Places the doorbell information into the socket internal queue.
4. If this was a blocking DIO socket the semaphore is posted.

5.4.5 Socket Receive

The following API [3] is used to receive data destined to a specific SRIO socket:-

```
Int32 Srio_sockRecv (Srio_SockHandle     srioSock,
                     Srio_DrvBuffer*     hDrvBuffer,
                     Srio_SockAddrInfo*  from)
```

As described above the received packet has already been placed into the socket internal queues. Thus the socket receive API simply checks if there is any data available in its internal queue and if so the received data is returned back to the application.

For non blocking sockets; if there is no data available on the socket the API will block till data is available. In the case of blocking sockets the API returns 0 if no data is available else it will return the number of bytes of data which is available.

For Normal mode sockets the API returns the SRIO Driver buffer returns the pointer to the data payload and for raw mode sockets the API returns the pointer to the buffer descriptor.

The application can now use the SRIO Driver buffer appropriately and once the application is done with the received buffer; it is the responsibility of the application to free up the received buffer. The following API [3] is used to achieve the needful

```
Void Srio_freeRxDrvBuffer (Srio_SockHandle srioSock,
                           Srio_DrvBuffer hDrvBuffer)
```

For Normal mode sockets; the data buffer is cleaned up by invoking the OSAL Memory cleanup API.

For RAW mode sockets the API calls the application registered call back function:-

```
void (*rawRxFreeDrvBuffer)(Srio_DrvBuffer hDrvBuffer);
```

This callback function is *specified while creating the application managed configuration*. This is done because for RAW sockets; the driver does not know the size of the descriptors etc and so it is the responsibility of the application to cleanup these descriptors by placing them into the appropriate return queues.

5.4.5.1 DIO

DIO sockets can only be used to receive doorbells.

In the case of blocking DIO sockets the API would check if there is any received doorbell in the socket internal queue. If there are no doorbells to be received the API would block by pending on a semaphore. The API would be woken up when a doorbell is received as explained in the DIO ISR handler.

In the case of non-blocking DIO sockets the API would check if there is any doorbell in the socket internal queue. If there is no doorbell the function would return immediately with a value of 0 indicating that there was no doorbell received.

A received doorbell is passed back to the application using the `hDrvBuffer` parameter in the `Srio_sockRecv` API. The received doorbell register and bit information can be extracted using the following macros:

```
#define SRIO_GET_DBELL_REG(DBELL_INFO)  CSL_FEXTR(DBELL_INFO, 31, 16)
#define SRIO_GET_DBELL_BIT(DBELL_INFO)  CSL_FEXTR(DBELL_INFO, 15,  0)
```

Before doorbells can be received by the socket the doorbell register and bit information need to be registered with the socket.

There is no need to perform any cleanup on the receive path for DIO sockets.

5.4.6 Set Runtime Configuration

The SRIO Driver exposes the following API [3] which is used to set the configuration parameters in the driver

```
Int32 Srio_setSockOpt(Srio_SockHandle srioSock,
                      Srio_optCommand optname,
                      Void* optval,
                      Int32 optlen)
```

The API provides a well defined standard entry point which can be used for configuring the various driver knobs.

Please refer to the SRIO driver documentation [3] for more information on the parameters required etc.

5.4.6.1 Packet Count

The option `Srio_Opt_PENDING_PKT_COUNT` can be used to configure the depth of the socket queues during run time by the application. The count is the maximum number of pending packets or doorbells which can be placed into the socket internal queue before they will get dropped.

5.4.6.2 Register Doorbell

The option `Srio_Opt_REGISTER_DOORBELL` can be used to register a socket with a specified doorbell register and bit. A socket can be registered with multiple doorbell register and bit information. But there can be only be 1 socket mapped to a specific doorbell register and bit. Once the mapping is created only then will the socket receive doorbells.

5.4.7 Get Runtime Configuration

The SRIO Driver also exposes the following API [3] which can be used to get the configuration parameters from the driver

```
Int32 Srio_getSockOpt(Srio_SockHandle srioSock,
                      Srio_optCommand optname,
                      Void* optval,
                      Int32 optlen)
```

Please refer to the SRIO driver documentation [3] for more information on the parameters required etc.

5.4.7.1 Packet Count

The option `Srio_Opt_PENDING_PKT_COUNT` can be used to get the depth of the socket queues during run time by the application.

5.4.7.2 Completion Code

The option `Srio_Opt_DIO_SOCK_COMP_CODE` is used to get the completion code for DIO non blocking sockets.

Please refer to the SRIO driver documentation [3] for more information on the parameters required etc.

5.4.8 Socket Close

Once the application is done with the socket; the following API [3] is used to close the socket

```
Int32 Srio_sockClose (Srio_SockHandle srioSock);
```

Once the socket has been closed it can no longer take part in any send and receive operations. The API cleans up internal memory associated with the sockets; flushes any received packets which were pending on the sockets internal queues & also removes the binding entries which map the socket to the SRIO IP Block.

5.5 OSAL

The OSAL is the operating system abstraction layer which is used to port the SRIO driver to a specific OS. The OSAL callouts are implemented in the "`srio_osal.h`" header file and need to be ported by the application developers to their specific operating system.

5.5.1 Memory Allocation

The SRIO Driver performs two types of memory allocations:

1. *Control Path Allocations*
   These allocations are typically during driver startup and initialization. The allocations here should return memory from the local memory.

2. *Data Path Allocations*
   In driver managed configuration; the driver allocates memory for receive and transmit buffers; these allocations fall into the fast path and such allocations should always be done into the global memory.

Each of the above type of allocations are defined into separate OSAL API; since this gives application developers the flexibility to differentiate between fast and slow path allocations and ensuring that they can plug more efficient memory allocation schemas into the fast path.

5.5.1.1 Control Path Memory Allocation

The SRIO Driver uses the `Srio_osalMalloc` macro to perform these allocations. The OSAL adaptation layer ports this macro to the following API prototype:-

```
Void* Osal_srioMalloc(UInt32 numBytes)
```

The parameter ‘`numBytes`’ reflects the total amount of memory that is requested. These allocations should be done only from the local memory space.

5.5.1.2 Data Path Memory Allocation

The SRIO Driver uses the `Srio_osalDataBufferMalloc` macro to perform these allocations. The OSAL adaptation layer ports this macro to the following API prototype:-

```
Void* Osal_srioDataBufferMalloc (UInt32 numBytes)
```

The parameter ‘`numBytes`’ reflects the total amount of memory that is requested. These allocations should be done only from the global memory space.

5.5.2 Memory cleanup

Depending on the type of allocation; the OSAL layer defines the corresponding memory cleanup API’s.

5.5.2.1 Control Path Memory Cleanup

The SRIO driver uses the `Srio_osalFree` macro to perform all memory cleanups for the control path. The OSAL adaptation layer ports this macro to the following API prototype:-

```
Void Osal_srioFree(Void* ptr, UInt32 numBytes)
```

The parameter ‘`ptr`’ reflects the address of the memory block which needs to be cleaned up. The parameter ‘`numBytes`’ reflects the size of the memory which is being cleaned up.

5.5.2.2 Data Path Memory Cleanup

The SRIO driver uses the `Srio_osalDataBufferFree` macro to perform all memory cleanups for the data path. The OSAL adaptation layer ports this macro to the following API prototype:-

```
Void Osal_srioDataBufferFree (Void* ptr, UInt32 numBytes)
```

The parameter ‘`ptr`’ reflects the address of the memory block which needs to be cleaned up. The parameter ‘`numBytes`’ reflects the size of the memory which is being cleaned up.

5.5.3 Logging API

Internally the SRIO driver uses the `Srio_osalLog` macro to perform all logging operations. The OSAL adaptation layer ports this macro to the following API prototype:-

```
void Osal_srioLog( String fmt, ... )
```

The parameter ‘`fmt`’ is a `printf` style formatted string. This should only be defined and used for debugging purposes.

5.5.4 Multicore Critical Section

The SRIO driver maintains certain data structures in shared memory to allow it to work across multiple cores. If the SRIO driver is operating in a mode where multiple cores are invoking the driver API then all accesses to this shared resource need to be protected and this is done through these API.

The SRIO driver exposes the `Srio_osalEnterMultipleCoreCriticalSection` macro to enter the Multicore critical section. The OSAL adaptation layer ports this macro to the following API prototype:-

```
void* Osal_srioEnterMultipleCoreCriticalSection(void)
```

The function returns an opaque handle which describes the critical section. The driver does not interpret this value and does no error checking.

The SRIO driver uses the `Srio_osalExitMultipleCoreCriticalSection` macro to exit the critical section. The OSAL adaptation layer ports this macro to the following API prototype:-

```
void Osal_srioExitMultipleCoreCriticalSection(void* critSectHandle)
```

The function passes back the opaque critical section handle.

5.5.5 Single Critical Section

The SRIO driver maintains certain per core specific data structures. These data structures need to be protected from access by multiple users running on the same core. Users are defined as entities in the system which uses the SRIO Driver API's. The critical section defined here should also take into account the context of these users (Thread or Interrupt) and define the critical sections appropriately.

For example: In a pure polling based multiple thread user scenario; since all the SRIO driver API's are being called from thread context the critical section API could be defined to use semaphore to provide protection.

However on an interrupt based system where SRIO Receive Interrupts are enabled through the accumulator the critical section API should be defined to disable and enable interrupts to provide protection.

The SRIO driver exposes the `Srio_osalEnterSingleCoreCriticalSection` macro to enter the single core critical section. The OSAL adaptation layer ports this macro to the following API prototype:-

```
void* Osal_srioEnterSingleCoreCriticalSection (Srio_DrvHandle drvHandle)
```

The function returns an opaque handle which describes the critical section. The driver does not interpret this value and does no error checking.

The function also takes as argument the SRIO Driver Instance handle. This is passed to handle the scenario where an application has opened multiple driver instances. *For example*: There is an application managed driver instance configured in polled mode and a driver managed driver instance configured in interrupt mode. If the application managed configuration is running in the same thread context then the single core protection for this instance should be a NOP. For the driver managed driver instance the single core protection could be to disable & enable interrupts. Exposing the driver instance in this call allows the application developers to select more efficient implementations depending upon the driver instance configuration.

The SRIO driver uses the `Srio_osalExitSingleCoreCriticalSection` macro to exit the critical section. The OSAL adaptation layer ports this macro to the following API prototype:-

```
void Osal_srioExitSingleCoreCriticalSection (Srio_DrvHandle drvHandle, void*
critSectHandle)
```

5.5.6 Memory Access Hooks

If the SRIO driver is running in a multi-core system use case; the access to the data structures in the shared memory need to be synchronized to ensure that the contents of the cache and memory are always in sync with each other.

The SRIO driver exposes the `Srio_osalBeginMemAccess` macro to indicate that an access to the specified memory region is starting. The OSAL adaptation layer ports this macro to the following API prototype:-

```
void  Osal_srioBeginMemAccess(void* ptr, uint32_t size)
```

The function takes as parameters the address of the memory & number of bytes which are being accessed. The memory if cached needs to be invalidated so that the contents of the cache are reloaded back from the actual memory. This will ensure that there is no stale data in the cache.

The SRIO driver exposes the `Srio_osalEndMemAccess` macro to indicate that an access to the specified memory region is ending. The OSAL adaptation layer ports this macro to the following API prototype:-

```
void  Osal_srioEndMemAccess(void* ptr, uint32_t size)
```

The function takes as parameters the address of the memory & number of bytes which are being accessed. The memory if cache needs to be written back so that the contents of the cache are synched up with the actual memory. This is true in the case of Write-back cache however if the caches are operating in Write through mode this API could be a NOP since the cache contents have already been written back to actual memory.

In addition to the above mentioned API the driver also exposes the following additional API for descriptors. This is to ensure that if the descriptors are located in Shared or external memory the cache contents can be synchronized before the descriptors are accessed.

```
void  Osal_srioBeginDescriptorAccess (Srio_DrvHandle drvHandle,void* ptr,
uint32_t descSize);

void  Osal_srioEndDescriptorAccess (Srio_DrvHandle drvHandle,void* ptr,
uint32_t descSize);
```

5.5.7 Blocking Sockets: OSAL Requirements

The section is applicable only for blocking sockets. Applications which decide not to use blocking API can stub out these functions.

5.5.7.1 Create Semaphore

All blocking sockets are associated with a semaphore. The semaphore is created during the socket open API. Internally the SRIO driver uses the `Srio_osalCreateSem` macro to perform this operation. The OSAL adaptation layer ports this macro to the following API prototype:-

```
Void* Osal_srioCreateSem(Void);
```

The function expects a critical section to be created and the handle to the created critical section is passed back to the SRIO driver. The SRIO driver attaches this semaphore to the internal socket structure and uses this for all other critical section operations.

Application developers should ensure that the semaphore created should be made ***unavailable*** during creation time.

5.5.7.2 Pend Semaphore

The application calls the SRIO driver receive API to read data from a specified socket. Now if there is no data available on the socket; the application cannot proceed further. A blocking socket would cause the application to "*wait for the semaphore*" at this time. The application thread which invoked the socket receive API would block here.

Internally the SRIO driver uses the `Srio_osalPendSem` macro to block on the semaphore. The OSAL adaptation layer ports this macro to the following API prototype:-

```
      Void Osal_srioPendSem(Void* semHandle);
```

The parameter '`semHandle`' is the semaphore handle which was created.

5.5.7.3 Post Semaphore

When data is received by the SRIO driver; the received data is mapped to a specific socket. The semaphore associated with this socket is then posted. Internally the SRIO driver uses the

`Srio_osalPostSem` macro to post the semaphore. The OSAL adaptation layer ports this macro to the following API prototype:-

```
        Void Osal_osalPostSem(Void* semHandle);
```

The parameter '`semHandle`' is the semaphore handle which was created and which is being posted.

Posting a semaphore causes any received applications which were blocked waiting for data to arrive to be woken up.

5.5.7.4 Delete Semaphore

When a socket is closed then the SRIO driver needs to ensure that any semaphore associated with the socket needs to be deleted too. Internally the SRIO driver uses the `Srio_osalDeleteSem` macro to delete the semaphore. The OSAL adaptation layer ports this macro to the following API prototype:-

```
        Void Osal_srioDeleteSem(Void* semHandle);
```

The parameter '`semHandle`' is the semaphore handle which was created.

5.6 Data Placement

The SRIO driver data structures can be broken up into the following categories:-

- Driver Instance specific
  These data structures are allocated & maintained per instance and these allocations are done from the local memory. The memory allocation API for this is as per the OSAL memory allocation API.

- Shared Memory
  These data structures are common & shared across multiple driver instances and need to be shared & available across different instances (across multiple cores). This data structure is relocated through the following `pragma` during linking time.

```
#pragma DATA_SECTION (gSRIODriverMCB, ".srioSharedMem");
```

If the SRIO driver is being executed on multiple cores i.e. driver instances are being opened and used on multiple cores then the data section `.srioSharedMem` should be relocated to shared memory.

However if the driver is being used only on a single core then the data section can be placed in local memory.

6 Integration

The SRIO driver depends on the following components:-

a. CPPI LLD
b. QMSS LLD
c. CSL

These components need to be installed before the SRIO driver can be integrated. The SRIO driver is released in source code and in pre-built library. Applications can decide how to use the SRIO driver.

The SRIO Driver release notes indicate the version of the above components which that release is dependent upon. The next steps use the version numbers for illustrative purpose only.

6.1 Pre-built approach

In this approach the application developers decide to use the SRIO driver pre-built libraries as is. The following steps need to be done:-

a. The application developers modify their application configuration file to use the SRIO package.

```
var Srio = xdc.loadPackage('ti.drv.srio');
```

b. Ensure that the XDCPATH is configured to have the path to the PDK package
c. This implies that XDC Configuration scripts will link the application using the SRIO Driver libraries (`Module.xs`)
d. The application authors need to provide an OSAL implementation file for SRIO and ensure that this linked with the application; failure to do so will results in linking errors. Please refer to the SRIO OSAL Section for more information on the API's which need to be provided.

Note: Since SRIO depends upon CPPI and QMSS the OSAL implementation should also have their implementations. Please refer to the CPPI and QMSS OSAL definitions for more information.

This approach is highlighted in the SRIO "`example`" projects.

6.2 Rebuild library

In this approach the application developers decide to use the SRIO driver source code and add these files to the application project to rebuild the SRIO driver code base. The following steps need to be redone:-

a. Application developers should port the file "`srio_osal.h`" to their operating system environment. *Developers are recommended to create a copy of this file and place it in their application directory.* They should use the file which is provided in the SRIO installation only as a template. The goal here should be to map the `Srio_osalXXX` macros to the OS calls directly thus reducing the overhead of an API callout. For example:

```
#define Srio_osalCreateSem()    (Void*)Semaphore_create(0, NULL, NULL)
```

b. Application developers should port the file "`srio_types.h`" to the application environment. *Developers are recommended to create a copy of this file and place it in their application directory.*

c. Append the include path to the top level SRIO package directory i.e. if the SRIO package is installed in `C:\Program Files\Texas Instruments\sriolld_c6498_1_0_0_1`; then make sure the include path is configured as `C:\Program Files\Texas Instruments\sriolld_c6498_1_0_0_1\packages`

d. Add the SRIO driver files listed in the `src` directory to the application build files

The approach above is highlighted in the SRIO `test` directory